How to catch an error in a linux bash script?

Asked 8 years, 7 months ago Modified 6 years, 5 months ago Viewed 145k times

15

7

I made the following script:

```
# !/bin/bash

# OUTPUT-COLORING
red='\e[0;31m'
green='\e[0;32m'
NC='\e[0m' # No Color

# FUNCTIONS
# directoryExists - Does the directory exist?
function directoryExists {
    cd $1
    if [ $? = 0 ]
            then
                    echo -e "${green}$1${NC}"
            else
                    echo -e "${red}$1${NC}"
    fi
}

# EXE
directoryExists "~/foobar"
directoryExists "/www/html/drupal"
```

The script works, but beside my echoes, there is also the output when

```
cd $1
```

fails on execution.

```
testscripts//test_labo3: line 11: cd: ~/foobar: No such file or directory
```

Is it possible to catch this?

bash shell shell-script error-handling

Share Improve this question

Follow

edited Oct 22, 2013 at 22:58

Gilles 'SO- stop being evil'

752k 180 1565 2054

asked Oct 22, 2013 at 10:29

user avata Thomas De Wilde

153 1 1 5

Just an FYI, you can also do this a lot simpler; `test -d /path/to/directory` (or `[[ -d /path/to/directory ]]` in bash) will tell you whether a given target is a directory or not, and it will do it quietly. – phemmer Oct 22, 2013 at 12:36

@Patrick, that just tests if it's a directory, not if you can `cd` into it. – Stéphane Chazelas Oct 22, 2013 at 12:54

@StephaneChazelas yes. The function name is `directoryExists` . – phemmer Oct 22, 2013 at 13:57

See a detailed answer here: Raise error in a Bash script. – codeforester Aug 26, 2018 at 19:41

5 Answers

Sorted by: Highest score (default)

15

Use `set -e` to set exit-on-error mode: if a simple command returns a nonzero status (indicating failure), the shell exits.

Beware that `set -e` doesn't always kick in. Commands in test positions are allowed to fail (e.g. `if failing_command` , `failing_command || fallback`). Commands in subshell only lead to exiting the subshell, not the parent: `set -e; (false); echo foo` displays `foo` .

Alternatively, or in addition, in bash (and ksh and zsh, but not plain sh), you can specify a command that's executed in case a command returns a nonzero status, with the `ERR` trap, e.g. `trap 'err=$?; echo >&2 "Exiting on error $err"; exit $err' ERR` . Note that in cases like `(false); …` , the ERR trap is executed in the subshell, so it can't cause the parent to exit.

Share Improve this answer Follow edited Jan 11, 2016 at 17:11

answered Oct 23, 2013 at 16:19

Gilles 'SO- stop being evil'

752k 180 1565 2054

Recently I experimented a little and discovered a convenient way of fixing `||` behavior, which enables to easily do proper error handling without using traps. See my answer. What do you think about that method? – skozin Jan 11, 2016 at 16:36

@sam.kozin I don't have time to review your answer in detail, it looks good on principle. Apart from portability, what are the benefits over ksh/bash/zsh's ERR trap? – Gilles 'SO- stop being evil' Jan 11, 2016 at 17:07

Probably the only benefit is composability, as you don't risk to overwrite another trap that was set before you function runs. Which is a useful feature when you're writing some common function that you will later source and use from other scripts. Another benefit might be full POSIX compatibility, though it is not so important as `ERR` pseudo-signal is supported in all major shells. Thanks for the review! =) – skozin Jan 11, 2016 at 17:20

1 @sam.kozin I forgot to write in my previous comment: you may want to post this on Code Review and post a link in the chatroom. – Gilles 'SO- stop being evil' Jan 11, 2016 at 17:24

Thanks for the suggestion, I'll try to follow it. Didn't know about Code Review. – skozin Jan 11, 2016 at 17:27

8

Your script changes directories as it runs, which means it won't work with a series of relative pathnames. You then commented later that you only wanted to check for directory existence, not the ability to use `cd`, so answers don't need to use `cd` at all. Revised. Using `tput` and colours from `man terminfo`:

```
#!/bin/bash -u
# OUTPUT-COLORING
red=$( tput setaf 1 )
green=$( tput setaf 2 )
NC=$( tput setaf 0 )      # or perhaps: tput sgr0

# FUNCTIONS
# directoryExists - Does the directory exist?
function directoryExists {
    # was: do the cd in a sub-shell so it doesn't change our own PWD
    # was: if errmsg=$( cd -- "$1" 2>&1 ) ; then
    if [ -d "$1" ] ; then
        # was: echo "${green}$1${NC}"
        printf "%s\n" "${green}$1${NC}"
    else
        # was: echo "${red}$1${NC}"
        printf "%s\n" "${red}$1${NC}"
        # was: optional: printf "%s\n" "${red}$1 -- $errmsg${NC}"
    fi
}
```

(Edited to use the more invulnerable `printf` instead of the problematic `echo` that might act on escape sequences in the text.)

Share Improve this answer Follow edited Oct 23, 2013 at 1:07

answered Oct 22, 2013 at 13:54

Ian D. Allen

797 4 11

That also fixes (unless xpg_echo is on) the issues when filenames contain backslash characters.
– Stéphane Chazelas Oct 22, 2013 at 14:28

8

To expand on the @Gilles' answer:

Indeed, `set -e` doesn't work inside commands if you use `||` operator after them, even if you run them in a subshell; e.g., this wouldn't work:

```
#!/bin/sh

# prints:
#
# --> outer
# --> inner
# ./so_1.sh: line 16: some_failed_command: command not found
# <-- inner
# <-- outer

set -e

outer() {
  echo '--> outer'
  (inner) || {
    exit_code=$?
    echo '--> cleanup'
    return $exit_code
  }
  echo '<-- outer'
}

inner() {
  set -e
  echo '--> inner'
  some_failed_command
  echo '<-- inner'
}

outer
```

But `||` operator is needed to prevent returning from the outer function before cleanup.

There is a little trick that can be used to fix this: run the inner command in background, and then immediately wait for it. The `wait` builtin will return the exit code of the inner command, and now you're using `||` after `wait`, not the inner function, so `set -e` works properly inside the latter:

```
#!/bin/sh

# prints:
#
# --> outer
# --> inner
# ./so_2.sh: line 27: some_failed_command: command not found
# --> cleanup

set -e
```

```
outer() {
  echo '--> outer'
  inner &
  wait $! || {
    exit_code=$?
    echo '--> cleanup'
    return $exit_code
  }
  echo '<-- outer'
}

inner() {
  set -e
  echo '--> inner'
  some_failed_command
  echo '<-- inner'
}

outer
```

Here is the generic function that builds upon this idea. It should work in all POSIX-compatible shells if you remove `local` keywords, i.e. replace all `local x=y` with just `x=y`:

```
# [CLEANUP=cleanup_cmd] run cmd [args...]
#
# `cmd` and `args...` A command to run and its arguments.
#
# `cleanup_cmd` A command that is called after cmd has exited,
# and gets passed the same arguments as cmd. Additionally, the
# following environment variables are available to that command:
#
# - `RUN_CMD` contains the `cmd` that was passed to `run`;
# - `RUN_EXIT_CODE` contains the exit code of the command.
#
# If `cleanup_cmd` is set, `run` will return the exit code of that
# command. Otherwise, it will return the exit code of `cmd`.
#
run() {
  local cmd="$1"; shift
  local exit_code=0

  local e_was_set=1; if ! is_shell_attribute_set e; then
    set -e
    e_was_set=0
  fi

  "$cmd" "$@" &

  wait $! || {
    exit_code=$?
  }

  if [ "$e_was_set" = 0 ] && is_shell_attribute_set e; then
    set +e
  fi

  if [ -n "$CLEANUP" ]; then
    RUN_CMD="$cmd" RUN_EXIT_CODE="$exit_code" "$CLEANUP" "$@"
    return $?
```

```
    return $?
  fi

  return $exit_code
}


is_shell_attribute_set() { # attribute, like "x"
  case "$-" in
    *"$1"*) return 0 ;;
    *)    return 1 ;;
  esac
}
```

Example of usage:

```
#!/bin/sh
set -e

# Source the file with the definition of `run` (previous code snippet).
# Alternatively, you may paste that code directly here and comment the next
line.
. ./utils.sh


main() {
  echo "--> main: $@"
  CLEANUP=cleanup run inner "$@"
  echo "<-- main"
}


inner() {
  echo "--> inner: $@"
  sleep 0.5; if [ "$1" = 'fail' ]; then
    oh_my_god_look_at_this
  fi
  echo "<-- inner"
}


cleanup() {
  echo "--> cleanup: $@"
  echo "    RUN_CMD = '$RUN_CMD'"
  echo "    RUN_EXIT_CODE = $RUN_EXIT_CODE"
  sleep 0.3
  echo '<-- cleanup'
  return $RUN_EXIT_CODE
}

main "$@"
```

Running the example:

```
$ ./so_3 fail; echo "exit code: $?"

--> main: fail
  > inner: fail
```

```
--> inner: fail
./so_3: line 15: oh_my_god_look_at_this: command not found
--> cleanup: fail
    RUN_CMD = 'inner'
    RUN_EXIT_CODE = 127
<-- cleanup
exit code: 127

$ ./so_3 pass; echo "exit code: $?"

--> main: pass
--> inner: pass
<-- inner
--> cleanup: pass
    RUN_CMD = 'inner'
    RUN_EXIT_CODE = 0
<-- cleanup
<-- main
exit code: 0
```

The only thing that you need to be aware of when using this method is that all modifications of Shell variables done from the command you pass to `run` will not propagate to the calling function, because the command runs in a subshell.

Share Improve this answer Follow

edited Apr 13, 2017 at 12:36
Community Bot
1

answered Jan 11, 2016 at 16:33

skozin
353 5 9

Apparently `set -e` doesn't work with `&&` after the function either, which is perhaps even more unexpected that it not working with `||` . – supersolver Dec 3, 2021 at 15:07

2

You don't say what exactly you mean by `catch` --- report and continue; abort further processing?

Since `cd` returns a non-zero status on failure, you could do:

```
cd -- "$1" && echo OK || echo NOT_OK
```

You could simply exit on failure:

```
cd -- "$1" || exit 1
```

Or, echo your own message and exit:

```
cd -- "$1" || { echo NOT_OK; exit 1; }
```

And/or suppress the error provided by `cd` on failure:

```
cd -- "$1" 2>/dev/null || exit 1
```

By standards, commands should put error messages on STDERR (file descriptor 2). Thus `2>/dev/null` says redirect STDERR to the "bit-bucket" known by `/dev/null`.

(don't forget to quote your variables and mark the end of options for `cd`).

Share Improve this answer Follow

edited Oct 22, 2013 at 12:52
Stéphane Chazelas
469k 83 924 1366

answered Oct 22, 2013 at 12:39
JRFerguson
13.7k 3 32 39

@Stephane Chazelas point of quoting and signaling end-of-options well taken. Thanks for editing. – JRFerguson Oct 22, 2013 at 13:36

Actually for your case I would say that the logic can be improved.

1

Instead of cd and then check if it exists, check if it exists then go into the directory.

```
if [ -d "$1" ]
then
     printf "${green}${NC}\\n" "$1"
     cd -- "$1"
else
     printf "${red}${NC}\\n" "$1"
fi
```

But if your purpose is to silence the possible errors then `cd -- "$1" 2>/dev/null` , but this will make you debug in the future harder. You can check the if testing flags at: Bash if documentation:

Share Improve this answer Follow

edited Jan 11, 2016 at 17:13

Gilles 'SO- stop being evil'

752k 180 1565 2054

answered Oct 22, 2013 at 10:52

BitsOfNix

4,907 2 24 34

This answer fails to quote the `$1` variable and will fail if that variable contains blanks or other shell metacharacters. It also fails to check whether the user has permission to `cd` into it. – Ian D. Allen Oct 22, 2013 at 13:38

I was actually trying to check if a certain directory existed, not necessarily cd to it. But because I didn't know better, I thought trying to cd to it would cause an error if not existed so why not catch it? I didn't know about the if [-d $1] that's exactly what I needed. So, thank you a lot! (I'm used to proram Java, and checking for a directory in an if statement is not exactly common in Java) – Thomas De Wilde Oct 22, 2013 at 18:30